A quick guide to reading, manipulating, plotting and writing data in CDAT

Outline

- Reading data from files
- Basic file/data manipulation
- Basic plotting
- Writing output to files

CDAT-compatible data formats (1)

But the best way to read data in CDAT is to use the "**cdms**" module. Recognised formats are:

- **NetCDF** (standard for input and output) – CDMS follows the Climate and Forecasts (CF) Metadata Convention for NetCDF.

- **HDF4** – currently incompatible with the NetCDF option due to library conflicts. CDAT can be built with either, not both. There is a hope ahead with a merger planned of NetCDF4 and HDF5 libraries (http://my.unidata.ucar.edu/content/software/netcdf/netcdf-4/index.html).

CDAT-compatible data formats (2)

- More recognised format are:
  - **GRIB** – is handled via the GrADS/GRIB interface, a slightly convoluted but effective way to get data into CDAT.
  - **PCMDI DRS** format – not covered here as relatively little UK usage.
  - **CDML** (Climate Data Markup Language) – the internal CDAT XML representation that points to multiple binary files.

Other self-describing formats of interest in the UK

- You can also get support for:
  - **PP-format** – the BADC has developed code for reading the Met Office proprietary field data format. This should soon be included in the I/O layer beneath CDMS (known as cdunif – a C-layer that provides read access to multiple formats, and write access to NetCDF). Ask for details.
  - **NASA Ames** – a group of ASCII formats developed at NASA for field experiments and data exchange. Used extensively in UK atmospheric research. The BADC has developed a Python package to bridge NASA Ames data into CDAT (http://home.badc.rl.ac.uk/astephens/software/nappy).

CDMS (The heart of CDAT!)

CDMS is the python package at the core of CDAT. It provides **the best way to read and write data:**

- Opening a file for reading:
  `>>> f=cdms.open(`*`file_name`*`)`
  - will open an existing file protected against writing
- Opening a new file for writing:
  `>>> f=cdms.open(`*`file_name`*`, ’w’)`
  - will create a new file even if it already exists
- Opening an existing file for writing:
  `>>> f=cdms.open(`*`file_name`*`, ’r+’) # or ‘a’`
  - will open an existing file ready for writing or reading

Reading data from a file

Multiple ways to retrieve data:

All of it:

```
>>> data=f('var')
```

Specifying dimension name and values:

```
>>> s=f('lnsp', time=("1999-1-1", "2000-12-31"), \
          level=1000, lon=5)
```

- can use *time*, *level*, *latitude*, *longitude*, *t*, *z*, *y*, *x*, *lat* and *lon*.
- can provide either two values in a tuple "()" or just one.
- times are strings whereas others are just values (int or float)

Or use "**`slice`**" and indices instead of values:

```
>>> s=f('tco3', lat=slice(index1,index2,step))
```

- "step" is useful if you want to get every n^{th} value in a dataset.

Interrogating a CDAT file/dataset

Before extracting data you can find out about the dataset or file with:

```
>>> f.id # returns the file/dataset name
>>> f.listvariables() # returns a list of variables in the file
>>> f.variables # is a dictionary of variables in the file
>>> f.axes # returns the axes in the file
>>> f.attributes # returns all the file attributes (including axes)
>>> f.getVariable('temp') # same as f('temp')
>>> f.listglobal() # returns a list of global file attributes
```

Remember: you can list the methods using “dir(<object>)”.

Interrogating the variable metadata (1)

- From your variable object you might want to find out:
  - What axes is this variable defined against?

```
>>> var.getAxisList() # to see all of them
>>> var.getLongitude() # longitude axis only
>>> var.getLongitude()[:] # longitude values
# var.getTime(), var.getLevel() – similar
>>> var.getGrid() # grid (if appropriate)
```

  - What shape is the variable?

```
>>> var.shape
```

  - What is the size (number of values) and rank of this variable?

```
>>> var.size()
>>> var.rank()
```

Interrogating the variable metadata (2)

- What is the missing value?

```
>>> var.getMissing()
```

- What attributes exist for this variable?

```
>>> var.listattributes()
```

- What is the value of attribute ‘name’?

```
>>> var.getattribute(‘name’) # = var.name
```

- What is the axis order of this variable?

```
>>> var.getOrder()
```

- What is all the metadata for this variable?

```
>>> var.attributes
```

Interrogating axes (1)

- From your axis object you might want to find out:
  - What does this axis look like?

```
>>> ax=var.getAxis(2)
>>> print ax
   id: latitude
   Designated a latitude axis.
   units:  degrees_north
   Length: 73
   First:  -90.0
   Last:   90.0
   Other axis attributes:
      axis: Y
   Python id:  40ba476c
```

Interrogating axes (2)

– What are the units?

```
>>> ax.units
```

– What are the actual values?

```
>>> ax.getValue() # or ax[:]
```

– Is it time? Is it latitude?

```
>>> ax.isTime() ; ax.isLatitude()
```

– What are the bounds (if they exist)?

```
>>> ax.getBounds()
```

– What is the key metadata for this axis?

```
>>> ax.listall()
```

– Is it a circular axis (i.e. longitude wraps around itself)?

```
>>> ax.isCircular()
```

Sub-setting and *squeezing* the actual data

- As we've already seen, when you want to subset data you can just specify the spatial and temporal region you want (and you can keep doing it…):

```
>>> import cdms
>>> f=cdms.open('file1.nc')
>>> var=f('temp', time=("1999-1", "1999-2"))
>>> slab1=var(level=16, latitude=(0, 90))
>>> slab2=slab1(latitude=(30,40))
>>> slab3=slab2(longitude=2)
# Note that you still have a 4-D variable,
# You might want to remove the singleton axes:
>>> slab4=slab3(squeeze=1)
# squeeze also comes in handy when plotting
```

Mathematical manipulation of data arrays

- Manipulating arrays (i.e. variables) is simple as the whole thing can be included in your equations:

```
>>> var4=(var1**0.5)+(var2/var3)

>>> var2=var1*2.5

>>> import MV.cos
>>> cosvar=MV.cos(var1)
```

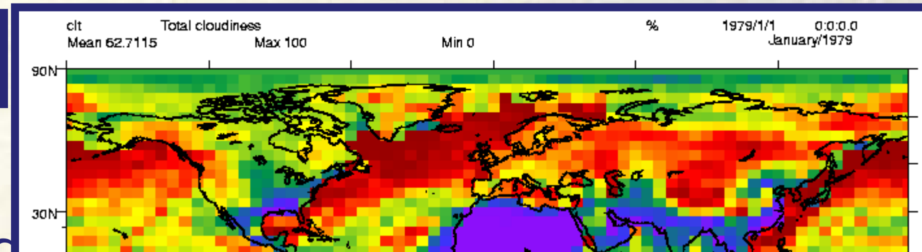
- Note: mathematical functions for arrays are in MV, for basic mathematical functions import the “math” module. E.g. `math.pi, math.cos` etc.

Creating simple plots with VCS

- All plotting requires the **VCS** module and a canvas to be created:

```
>>> x.plot(2Dfield)
```

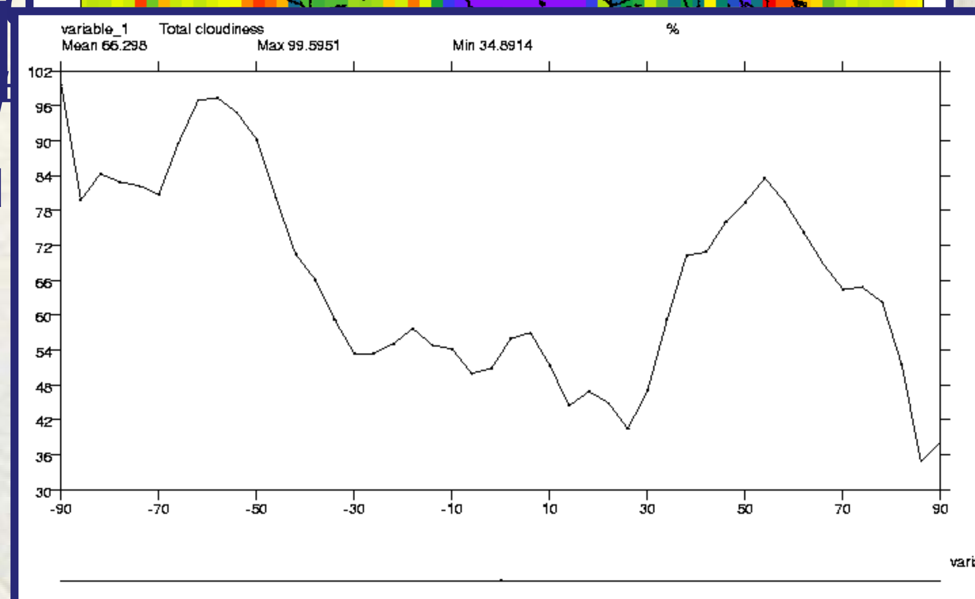
```
>>> x=vcs.init()
```



- You then use method

```
>>> x.plot(2Dfield)
```

```
>>> x.plot(data) #
>>> x.plot(2Dfield
>>> x.plot(1Ddata)
```



- Note: for 3D (or 4D) field and plot that.

Saving a VCS plot

- Once you have created a plot you can save it in one of various formats, examples are:

```
>>> x.plot(data)
>>> x.gif("myfile.gif")  # writes a GIF file

>>> x.ps("mypostscript.ps")  # writes a PS file
```

Opening a file for writing

- To write a new CDMS file:

```
>>> outfile=cdms.open('myfile.nc', 'w')
>>> # and to close:
>>> outfile.close()
```

- Note: *Data may not be written to a file until you close it, so make sure you do!*

- Same grammar as the built-in open function! This can be a reason to not import everything from CDMS because “ **from cdms import *** ” will overload the built-in ‘open’ function.

Writing file variables and attributes

- Writing CDMS variables, Numeric arrays or Masked Arrays to a CDMS file object is very easy:

```
>>> outfile.write(myvar)
>>> outfile.write(a_numeric_array)
```

- Writing file attributes (file level metadata) corresponds to setting global attributes in a NetCDF file and is simply done by setting class attributes:

```
>>> outfile.source="Data from Galaxy 4B02"
>>> outfile.sauce="Ketchup"
>>> outfile.version="3.1"
```

Basic File I/O example

- File I/O to NetCDF is simple:

```
import cdms
ufile = cdms.open('u_wind.nc')
vfile = cdms.open('v_wind.nc')

u = ufile('u')
v = vfile('v')

wind_speed = (u**2 + v**2)**0.5
outfile = cdms.open('wspd.nc', 'w')
outfile.write(wind_speed)
outfile.close()
```

← cdms.open function binds ufile to an instance of *CdmsFile*

← u and v are instances of the *TransientVariable* class.

← wind_speed is a new *TransientVariable* instance

← outfile is another *CdmsFile* instance with write permission